```yaml
version: "3"

services:

  client:
    build:
      context: ./client
      dockerfile: Dockerfile
    ports:
      - 5173:5173
    volumes:
      - ./client:/client
      - /node_modules
    environment:
      - CHOKIDAR_USEPOLLING=true


  server:
    build:
      context: ./server
      dockerfile: Dockerfile
    ports:
      - 3000:3000
    volumes:
      - ./server:/server
      - /node_modules
    depends_on:
      postgres:
        condition: service_healthy



  postgres:
    image: postgres:14
    environment:
      POSTGRES_PASSWORD: theta
      POSTGRES_USER: theta
      POSTGRES_DB: theta
    volumes:
      - ./pgdata:/var/lib/postgresql/data
    ports:
      - '5432:5432'
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U theta"]
      interval: 5s
      timeout: 5s
      retries: 5
```

- To start the application, we use Docker-Compose, this builds our Client & Server Dockerfile and mounts their container folders onto our local disk to enable live-reload during development. Also, all URLs and DB Config info have been hardcoded throughout the application.

- The server waits for a healthy DB before starting to ensure it can connect when the DB is ready to receive connections. I also populated some data on startup.

```js
const db = {};
db.Sequelize = Sequelize;
db.employee = require("./employee.model.js")(sequelize,Sequelize);
db.department = require("./department.model.js")(sequelize,Sequelize);
db.position = require("./position.model.js")(sequelize,Sequelize);
db.department.hasMany(db.employee, {
  foreignKey: {
    allowNull: false,
  },
})
db.position.hasMany(db.employee, {
  foreignKey: {
    allowNull: false,
  },
})
db.employee.belongsTo(db.department)
db.employee.belongsTo(db.position)
```

- Our Server is using Sequelize ORM to interact with Postgres, ORMs are helpful because they provide layers of abstraction to interact with the DB with Programming Language functions and provide safety to prevent things like SQL injections. React also messes around with Types so much, I'm glad the ORM handles boolean<>string<>number type conversion.

- Our Database Schema contains 3 Tables: Employees, Departments, and Positions. Many Employees can belong to Departments and Positions. I split up the Departments and Positions into tables to make it easier to Add and Remove new Departments/Positions in the Future. Some considerations, I wasn't sure whether to allow null relationships for these values, just incase one of those relations does get deleted. I also didn't test the cascading for when one of those relations gets deleted.

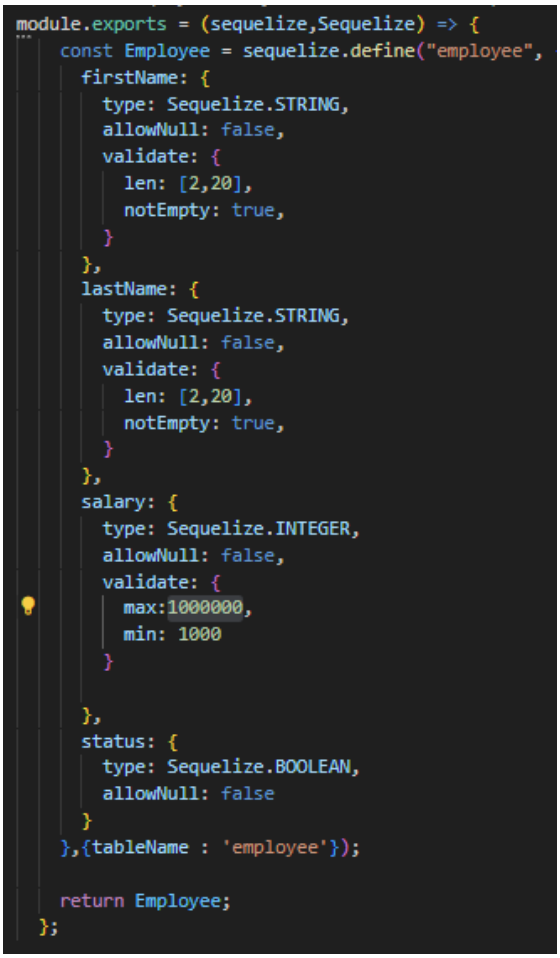
```js
var express = require('express');
var router = express.Router();
var apicache = require('apicache')
var cache = apicache.middleware

const employee = require("../controllers/employee.controller")

/* GET employees. */
router.get('/', cache('1 hour'), employee.findAll);
router.post('/', employee.create);
router.put('/', employee.update);
router.delete('/', employee.delete);

module.exports = router;
```

- We'll focus on the Employee routes, the Department & Position routes just have one GET ALL route.

- The Employee GET ALL route allows the client to get all Employees, also allowing for Request Params to be passed in to only retrieve all Employees with a

given Department or Status. Filtering is done on the Backend because Employee records can become large, due to this Caching & Pagination have also been implemented on this route. Page Size is hardcoded to 10.

- The POST(Create), PUT(Update), & DELETE routes only interact with a Single Employee Record at a time, with their data existing in the Request Body. These routes also Clear the Cache.



```
module.exports = (sequelize,Sequelize) => {
    const Employee = sequelize.define("employee",
      firstName: {
        type: Sequelize.STRING,
        allowNull: false,
        validate: {
          len: [2,20],
          notEmpty: true,
        }
      },
      lastName: {
        type: Sequelize.STRING,
        allowNull: false,
        validate: {
          len: [2,20],
          notEmpty: true,
        }
      },
      salary: {
        type: Sequelize.INTEGER,
        allowNull: false,
        validate: {
          max:1000000,
          min: 1000
        }

      },
      status: {
        type: Sequelize.BOOLEAN,
        allowNull: false
      }
    },{tableName : 'employee'});

    return Employee;
  };
```

- The Entities are broken up into three folders, their Model (shown left), that defines the schema along with Validation.
- A Controller that defines the logic for CRUD operations
- A Route that exposes that Controller to the Client.

Now We'll Move onto the Frontend on the Next Page

```tsx
const [isLoggedIn, setIsLoggedIn] = useState(false)

const [employeeResponse, setEmployeeResponse] = useState<EmployeeResponse | undefined>(undefined)
const [page, setPage] = useState<number>(0)

const [departments, setDepartments] = useState<any | undefined>([])
const [positions, setPositions] = useState<any | undefined>([])


const [departmentFilter, setDepartmentFilter] = useState<string>("")
const [statusFilter, setStatusFilter] = useState<string>("")

const [employeeAction, setEmployeeAction] = useState<string | undefined>(undefined)
const [selectedEmployee, setSelectedEmployee] = useState<any>({})

const [refreshState, setRefreshState] = useState(1)

const defaultForm = {
  firstName: '',
  lastName: '',
  salary: '',
  status: 'true',
  departmentId: 1,
  positionId: 1
}

const [formState, setFormState] = useState<any>(defaultForm)
useEffect(() => {

  if(isLoggedIn) {

    (async () => {
      const employeeData = await getEmployees(departmentFilter,statusFilter,page)
      console.log(employeeData)
      setEmployeeResponse(employeeData)

      const departments = await getDepartments()
      setDepartments(departments)
      setFormState((prevState:any) => ({
        ...prevState,
        departmentId: departments[0].id
    }));

      const positions = await getPositions()
      setPositions(positions)
      setFormState((prevState:any) => ({
        ...prevState,
        positionId: positions[0].id
    }));

    })()
  }
  else{
    setEmployeeResponse(undefined)
    setDepartments([])
    setPositions([])
    setDepartmentFilter("")
    setStatusFilter("")
    setPage(0)
  }

},[departmentFilter,statusFilter,page,isLoggedIn,refreshState])
```

- The frontend is extremely messy, I threw everything into the App.tsx. Not happy with the styling at all. I used Bulma for the CSS Framework.
- Anyway, let's start with our Application state.

- The isLoggedIn state is just a trigger for the application to get a go ahead to start retrieving data from the Backend.

- I start off with using TypeScript types to typecast the incoming Employee data, but I drop strict typing somewhere along development for the sake of time.

- The Page state refreshes the GET ALL Employees Backend call to get the next Pagination data if it exists.

- We get our Departments and Positions to populate Filter & Form 'select' fields.

- The DepartmentFilter & StatusFilter also refresh the GET ALL Employees Backend call when a Filter is selected.

    - EmployeeAction & SelectedEmployee are related to the Add & Edit Employee Modal. EmployeeAction determines whether the Modal is Adding or Editing the Employee and the SelectedEmployee is used to check against the FormState to determine whether the Modifications are different to allow the Frontend to let the User update the Employee.
- The RefreshState is a random number that just gets updated after an Employee is Added, Edited, or Deleted to tell the Application to make another GET ALL Backend call.
- You can see the states that cause a Backend Refresh at the end of the "useEffect" dependency list.

- We have a custom getConfirmation & Confirmation stateHook that opens a Modal that React Awaits a Response from to determine whether or not to delete an Employee.

```
function isValidName(name: any){
  return name && name.length >= 2 && name.length <= 20
}

function isValidSalary(salary: any){
  return salary && salary >= 1000 && salary <= 1000000
}

function isValidForm(){
  return isValidName(formState.firstName) && isValidName(formState.lastName) && isValidSalary(formState.salary)
}
```

```
    {employeeAction == "add" && <button className={isValidForm() ? 'button is-success' : "button is-dark"} disabled={!isValidForm()}
                                 onClick={async () => {
                                   await addEmployee(formState)
                                   setRefreshState(refreshState+1)
                                   setFormState(defaultForm)
                                   setEmployeeAction(undefined)
                                 }}>Save</button>}
    {employeeAction == "edit" && <button className={isValidForm() && !isEqualWith(formState,selectedEmployee,cmpStr2Num) ? 'button is-success' : "button is-dark"}
    disabled={!isValidForm() || isEqualWith(formState,selectedEmployee,cmpStr2Num)}
                                 onClick={async () => {
                                   await updateEmployee(formState)
                                   setRefreshState(refreshState+1)
                                   setFormState(defaultForm)
                                   setEmployeeAction(undefined)
                                 }}>Update</button>}
    {employeeAction == "edit" && isValidForm() && isEqualWith(formState,selectedEmployee,cmpStr2Num) && <p className='has-text-warning'> Make Changes To Update
    Employee</p>}

   </div>

 </div>
 <button onClick={() => {
   setFormState(defaultForm)
   setEmployeeAction(undefined)
 }} className="modal-close is-large" aria-label="close"></button>
```

- Here we have some Validation checks for the Frontend side to prevent a User from submitting invalid data. I wouldn't prefer relying on React State for Form data, I would use a combination of Refs or another Library to handle it more thoroughly. Since you can't control the Asynchronicity of React State, I feel like there could be race conditions where a User could submit invalid forms by accident if they're typing too fast or something.
- To maintain responsiveness and consistency of the Modal, the FormState is wiped after closing the Modal or after Adding or Updating an Employee.

Closing

That is the main bulk of my design and design decisions. Things I would improve with more time would definitely be styling, more error handling, null check considerations, more parameterization, state cleanup, and edge case testing. Thanks for reading.